Multithreading for Linear Algebra in Distributed Memory Environments

Parry Husbands

Interactive Supercomputing

[Joint work with Esmond Ng (LBNL) and Katherine Yelick (LBNL/UCB)]

Our Road to Multithreading

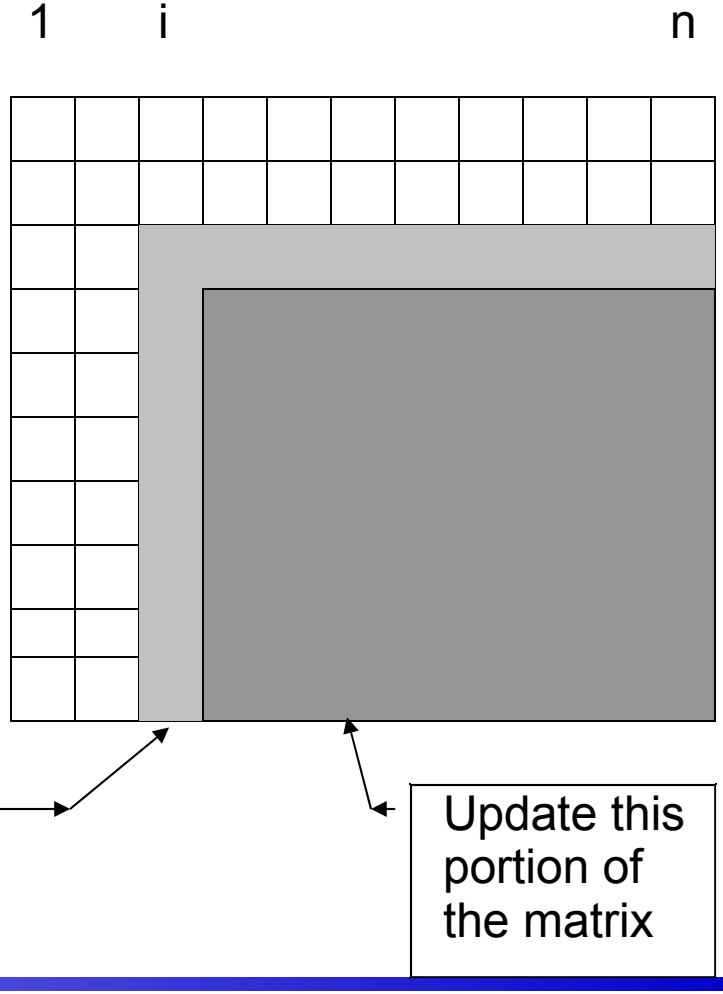
- Distributed Memory programming challenges
  - Expressibility
    - Many algorithmic constructs tortuous to implement
  - Performance
    - Synchronous codes spend an excessive amount of time waiting
- Asynchronous memory operations boost performance
  - Modern out-of-order processors
  - MPI_Isend()/MPI_Irecv()
- How do we organize programs with many outstanding requests?
  - Threads have a natural latency tolerance for both algorithmic and communication latencies
- Write distributed memory code in a multithreaded style!

LU Factorization with Partial Pivoting

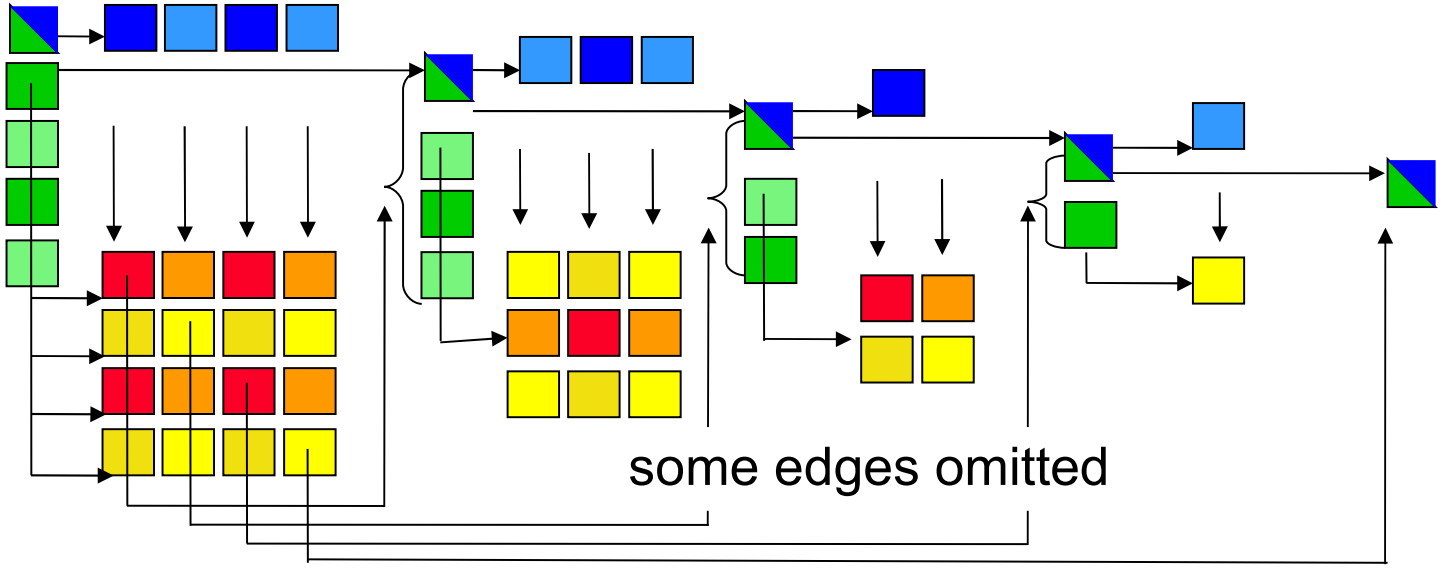
- A simple but heavily used computational kernel.
- Available in Linpack/LAPACK/ScaLAPACK.
  - LAPACK/ScaLAPACK are the second top mathematical libraries at the National Energy Research Scientific Computing Center, a national high performance computing facility funded by the Office of Science in the U.S. Department of Energy.
- HPL benchmark.
  - Highly tuned parallel block LU factorization with partial pivoting.

LU Factorization with Partial Pivoting (2)

```
for i=1:n-1
    swap rows so |a(i,i)| = max{abs(a(:,i))}
    for j=i+1:n
        l(j,i) = a(j,i)/a(i,i)
    for j=i:n
        u(i,j)=a(i,j)
    for j=i+1:n
        for k=i+1:n
            a(j,k) = a(j,k)-l(j,i)*u(i,k)
```



Parallel Tasks in LU



- Panel Factorizations (parallel recursive formulation used)
- Pivot application and update to U
- Trailing matrix updates

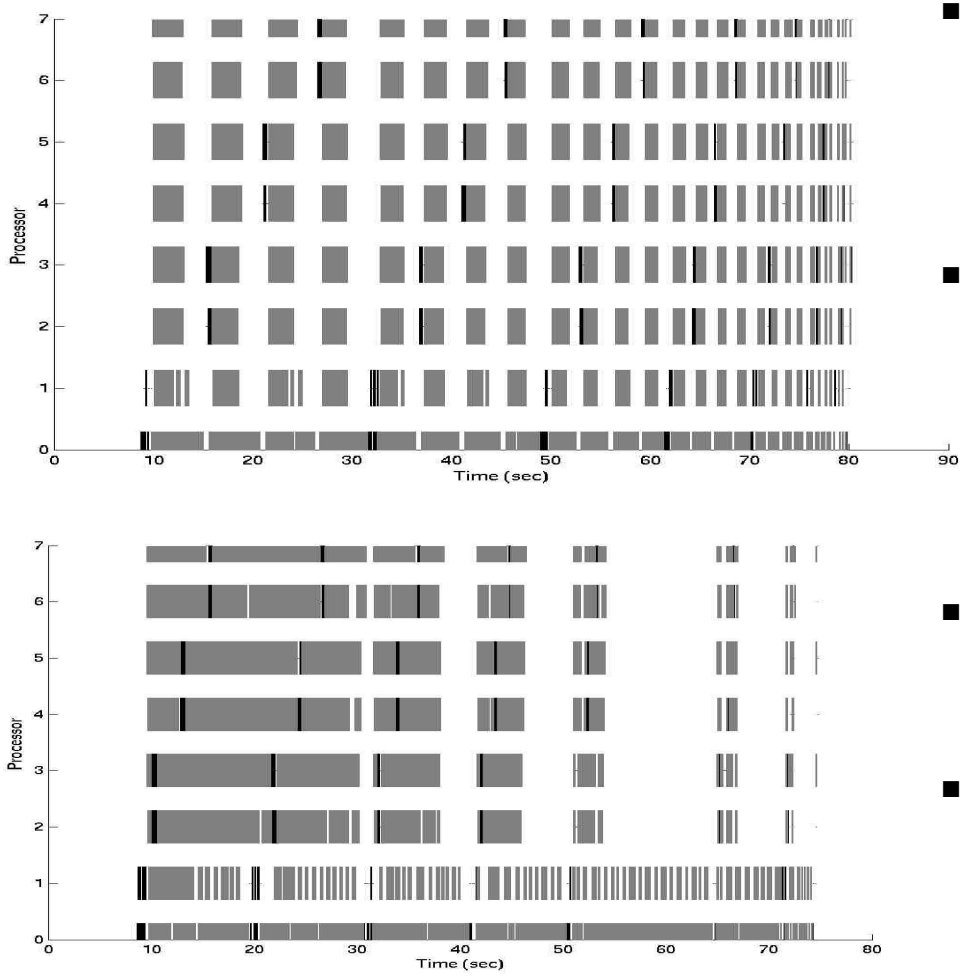
Distributed Memory Multithreading with UPC

- Co-operative multi-threading used to mask latency and to mask dependence delays (home-grown package)
- Non-blocking (remote get) transfers to mask communication latency
- Remote enqueue used to spawn remote threads. Threads are placed to take advantage of locality
- Matrix blocks distributed in 2-d block-cyclic manner (fixed layout) and tuned for block size
- Three levels of threads:
  - UPC threads (data layout, each runs an event scheduling loop)
  - Multithreaded BLAS (boost efficiency)
  - User level (non-preemptive) threads with explicit yield
- Operations "fire" when dependencies are satisfied (use a per proc. scoreboard). "Lookahead" is therefore dynamic (as in many shared mem. codes)

The Threads

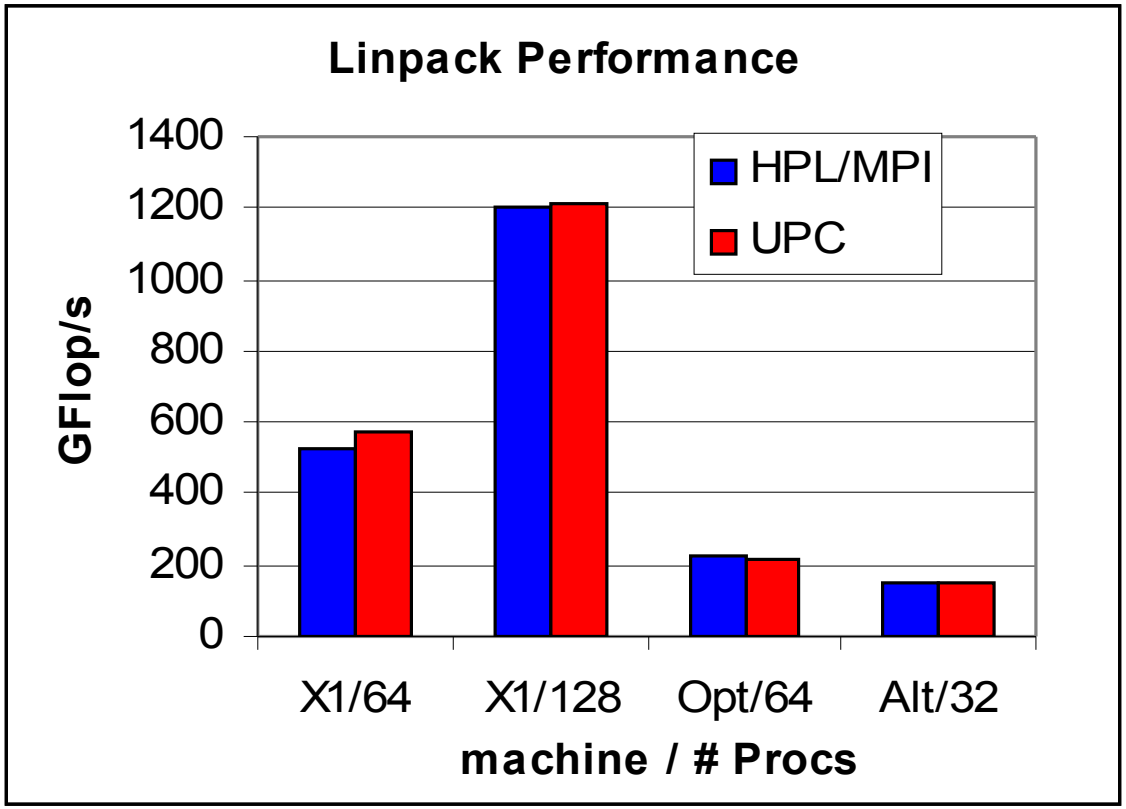
- Co-operative threads
  - Remove need to maintain integrity of data structures throughout program
  - Experimented with GNU Pth, POSIX Threads,
    Hand rolled user-level threads for portability
    - Uses only function calls and returns (fast context switches)
    - "Interesting" use of Duff's Device
    - Macros: PTP_SPAWN, PTP_FUNCALL, PTP_YIELD,
      PTP_START, PTP_END
    - Suspend, resume, priorities
    - Custom script expands, computes jumps, rewrites local (stack) accesses, creates functions for arguments, etc.
    - Allows for many threads to be created/destroyed per processor

Utilization Comparison



- Synchronous (above) vs. asynchronous (below) schedule
- SGI Altix Itanium 2 1.4GHz, n=12,800, process grid = 2x4, block size = 400
- Grey blocks = matrix multiplication
- Black blocks = panel factorization

UPC HP Linpack Performance

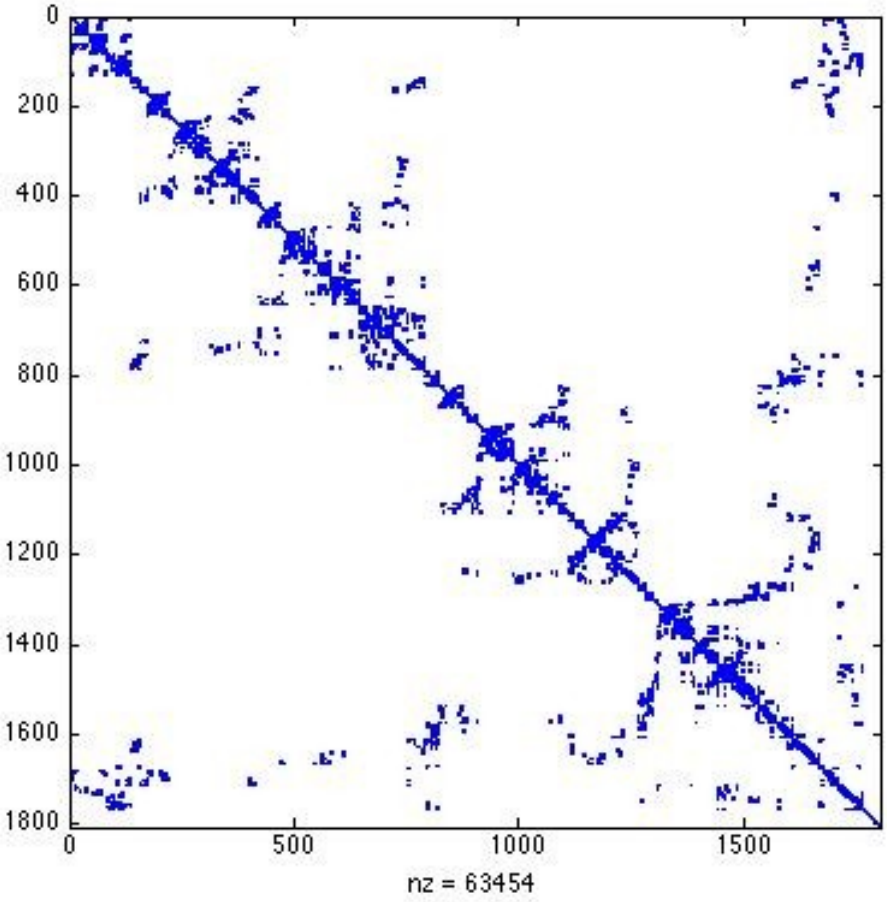


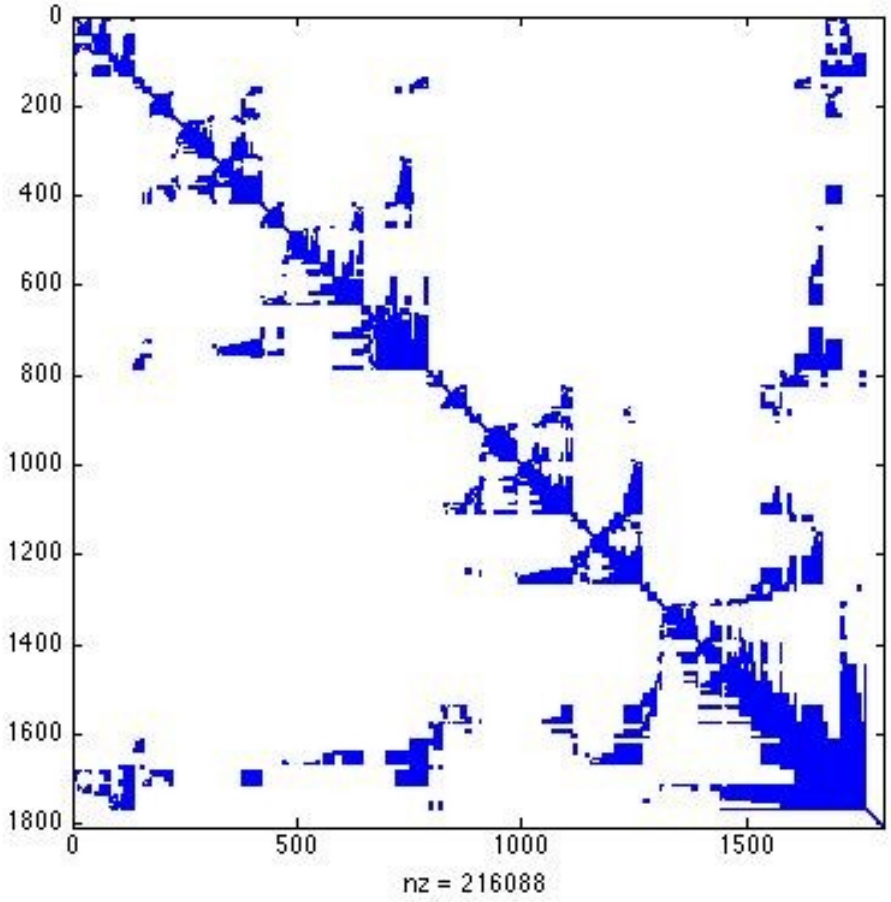
- Faster than ScaLAPACK (less synchronization), comparable to MPI/HPL
- Large scaling of UPC code on Itanium/Quadrics (Thunder)
  - 2.2 TFlops on 512p and 4.4 TFlops on 1024p
  - 91.8% of peak on 1p Itanium 2 1.5GHz, 81.9% on 1p Opteron 2.2GHz

Scheduling: The Major Issue

- Critical operation: Panel Factorization
  - need to satisfy its dependencies first
  - perform trailing matrix updates with low block numbers first
    - Use a Priority Queue to schedule these
  - panel factorizations started as soon as blocks of next panel are ready
- Theoretical and practical problem: Memory utilization
  - Not enough memory for all tasks at once. (Each update needs two temporary blocks, one from L, one from U)
  - If updates are scheduled too soon, you will run out of memory
  - Allocate memory in increasing order of factorization and don't skip any!
  - Thread blocks until enough memory available
- Cache performance: Too many dgemms to worry about the cache

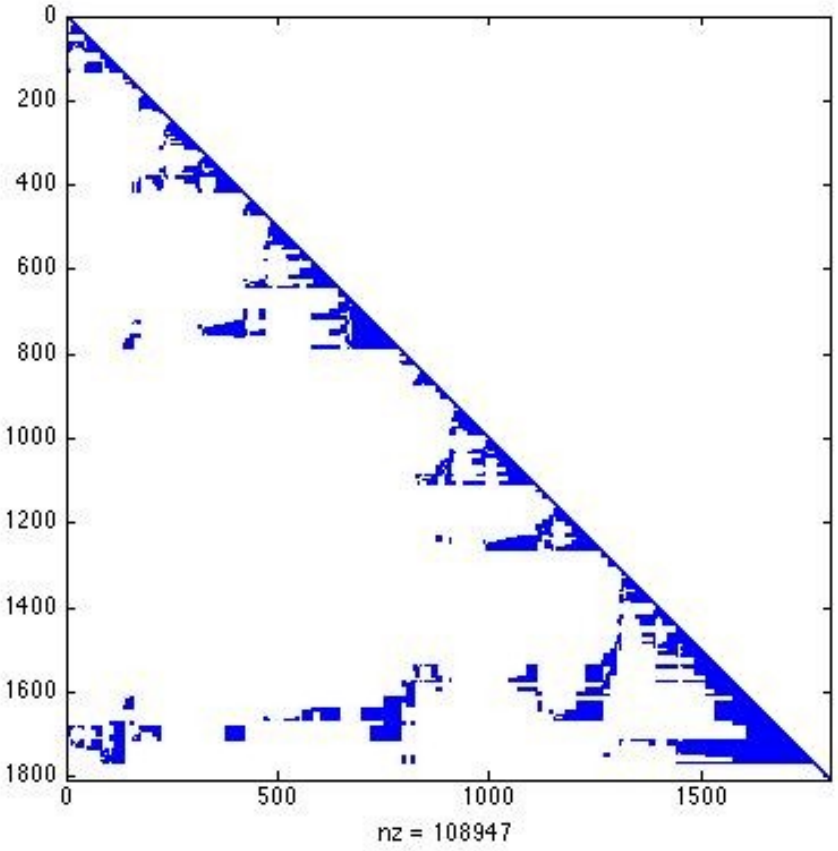
Sparse Matrix Factorization





Sparse Matrix Factorization

- Same basic algorithms used … but
- For efficiency we must take care to avoid operating on as many zero elements as possible
- Many variants due to symmetry, different orderings of basic factorization loop (left-looking, right-looking, multifrontal)
- High degree of parallelism (due to sparsity), but finer-grained (due to fewer nonzero elements)



Sparse Cholesky Factorization

- Based on left-looking, blocked serial code of Ng and Peyton
  - Choice of blocks to enhance performance via level-3 BLAS operations
  - Block columns receive updates from earlier block columns
  - After all updates are received, a block column is factorized
- Complications
  - Dependency graph
  - Scoreboard no longer simple
  - How do we choose the "best" operation to perform?
    - Longest path in chain of dependencies?
    - Weight this by amount of work?

Our Multithreaded Implementation

- Strategy
  - Use analysis to figure out dependencies and importance of each update
  - Threads for block column-block column updates
  - Set thread priorities based on importance

  Critical operations scheduled based on dependency graph
  Memory utilization controlled by performing critical ops first.
  Cache: What's a good schedule for this?

Preliminary Cholesky Performance

- Results obtained in SGI Altix (1.4GHz Itanium 2)
- Performance in seconds

	bmw7st_1	bmwcra_1	bmw3_2
n	141,347	148,770	227,362
nnz	3,740,507	5,396,386	5,757,996
1p	11.21	51.80	23.27
2p	6.97	30.00	12.69
4p	4.58	15.72	9.10
8p	2.73	8.52	5.31
sequential	7.21	34.61	15.59

- But… 1p performance not competitive with original serial version! So back to the drawing board…

Conclusion and Open Questions

- Portable addition of cooperative threads and remote function invocation to UPC
- High performance UPC version of Linpack Benchmark in ~5K LOC
- Sparse Cholesky still has issues
  - Need more thinking about scheduling

- Remember the scheduler's influence on
  - Critical tasks
  - Memory
  - Cache

Extras

Asynchronous Implementations

- MPI
  - Use non-blocking communication primitives
    - MPI_ISend()/MPI_IRecv()/MPI_IBcast()
  - Poll for incoming messages then perform work
- Multithreaded languages (PThreads, Cilk, …)
  - Use threads for each major operation
    - Each thread is a computational task that shares the CPU with other such tasks
  - Thread synchronization primitives manage algorithm dependencies
    - Give up the CPU (yield) to another thread when a long-latency network call is made
    - Suspend and resume other threads that may interfere with current work

Parallel Performance

- SGI Altix

- 8 procs (2 x 4 grid, n = 25,600)
  - ScaLAPACK (synchronous)
    25.25 GFlop/s (best block size 64)
  - UPC LU (asynchronous)
    33.60 GFlop/s (best block size 256)
  - 33% increase in performance
- 16 procs (4 x 4 grid, n = 32,000)
  - ScaLAPACK (synchronous)
    43.34 GFlop/s (block size 64)
  - UPC LU (asynchronous)
    70.26 Gflop/s (block size 200)
  - 62% increase in performance

Communication Requirements

- Processors usually arranged in a 2D grid.
- Reductions (finding the maximum in a distributed column) for pivot selection.
  - A gather operation.
- Row Exchanges for application of pivot sequence.
- Row Broadcasts for
  - Trailing matrix updates.
  - Updates to U.
- Column Broadcasts for trailing matrix updates.

Some Open CS Issues

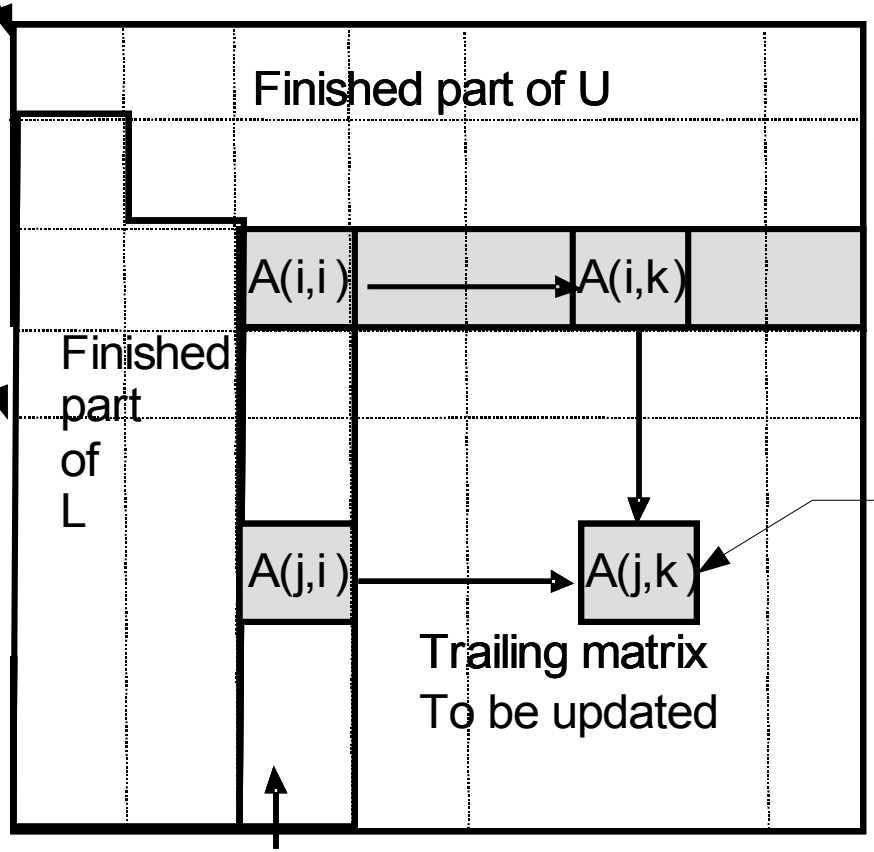
Future Investigations:

- How do things change with pre-emptive threads?
- Can we get support for remote enqueue and spawning?
- How to exert control over the local schedule in a principled way?
- Deadlock avoidance in resource allocation?

HPL (Parallel Block LU Factorization)

Matrix decomposed into blocks

Blocks 2D block-cyclic distributed for load balancing



Part of U to be updated.

Panel factorizations involve communication for pivoting

Matrix-matrix multiplication used here. Can be coalesced

Synchronous vs. Asynchronous Codes

- Synchronous codes
  - Pause other processors during panel factorization
  - Wait until trailing matrix update is complete before starting next factorization
  - Less performance
  - Easier to write

Synchronous vs. Asynchronous Codes

- Asynchronous codes
  - Exploit overlap - do something useful while waiting for data
  - Panel factorization can start as soon as data is ready
  - Trailing matrix updates overlapped with factorizations and other updates
  - Peak performance
  - Harder to write
  - Networking technology, infrastructure not always there

    J.B. White & S.W. Bova. "Where's the overlap?" (1999).